SE448: Information and Cyber Security	**Recitation 5**
Semester 2 5785	**24 April 2025**
Lecturer: Michael J. May	**Kinneret College**

Merkle Trees with the crums-io Library

1 Introduction to Merkle Trees

In the realm of computer security and data integrity, ensuring that data remains untampered and verifiable is paramount. A powerful data structure that addresses this need is the **Merkle Tree**, also known as a hash tree. It's a tree-like structure that efficiently summarizes the integrity of a large set of data.

At the heart of a Merkle Tree lies cryptographic hashing. Each leaf node in the tree represents the hash of a data block. Non-leaf nodes, on the other hand, contain the hash of the concatenation of their child nodes' hashes. The hierarchical hashing continues until it reaches the root of the tree, known as the **Merkle Root**. A sample Merkle tree is shown in Figure 1

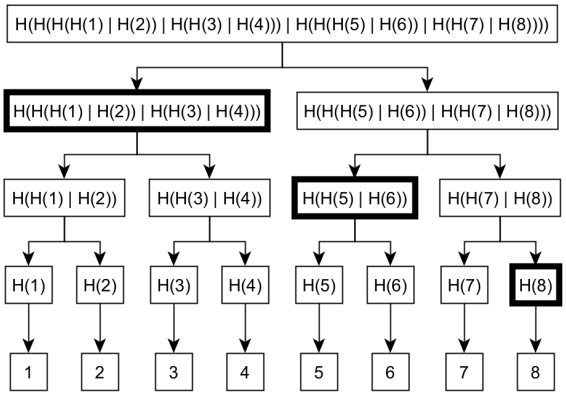

Figure 1: Merkle tree with 8 leaf nodes. H(x) is the application of hash function H to input x and — is the concatenation operator.

The beauty of a Merkle Tree lies in its ability to provide efficient and secure verification of data integrity. By comparing the calculated Merkle Root of a received dataset with a trusted Merkle Root, one can quickly determine if any part of the data has been modified. Furthermore, Merkle Trees enable **proof of inclusion**, allowing a user to verify that a specific data block is part of the original dataset without needing to access the entire dataset. This is achieved by providing a **Merkle Proof**, which consists of the hashes of the sibling nodes along the path from the leaf node to the root.

For instance, in the Merkle tree shown in Figure 1, membership of 7 in the tree can be proven by providing the 3 digests shown with a thick border to a verifier who has a trusted copy of the root digest.

Merkle Trees find widespread applications in various fields, including:

Cryptocurrencies Ensuring the integrity of transaction history (e.g., in Bitcoin).

Version Control Systems Detecting changes in files (e.g., in Git).

Distributed Systems Verifying data consistency across multiple nodes.

Certificate Authorities Providing efficient revocation checks.

2 The crums-io Merkle Tree Library

We will use the **crums-io** library, a Java library that provides a straightforward and efficient implementation of Merkle Trees. The library simplifies the process of creating, manipulating, and verifying Merkle Trees within your Java applications.

To use the crums-io library, you will need to include it as a dependency in your Java project (e.g., using Gradle). The library offers a well-structured API that allows you to easily perform common Merkle Tree operations. You can see the documentation for the library at:

- `https://crums-io.github.io/merkle-tree/`
- `https://github.com/crums-io/merkle-tree`

2.1 Important Functions

The crums-io library provides several key functions for working with Merkle Trees. Here are some of the most important ones you will use:

Builder The class provides a builder pattern to construct a new Merkle Tree. You will use methods of this builder to add data (leaves) to the tree.

Builder.add(byte[data)] The method adds a data block (as a `byte[]`) to the Merkle Tree as a leaf node. The library will automatically hash the data.

Builder.build() The method finalizes the construction of the Merkle Tree and returns a `Tree` object.

Tree.hash() Returns the Merkle Root of the constructed tree as a `byte[]`. This is the crucial value for verifying the integrity of the entire dataset.

Tree.idx().count() Returns the total number of leaf nodes (data nodes) in the tree.

Tree.proof(int leafIndex) Returns a `Proof` about the inclusion of the given data block in the tree. The method generates a Merkle Proof. The proof is a list of byte arrays representing the sibling hashes required to verify the inclusion of the data.

Proof.verify(MessageDigest) Verifies that a proof is valid using the provided `MessageDigest` algorithm.

Proof.rootHash() Returns the root hash associated with a given `Proof`. Check that the root hash matches what you expected when verifying!

2.2 Basic Merkle Tree Operations

The fundamental operations you will perform with the crums-io library include:

1. Creation: Building a Merkle Tree from a set of data blocks. This involves adding each data block as a leaf and then recursively hashing up the tree.
2. Verification: Checking if a specific data block is part of the Merkle Tree and if it has not been tampered with. This involves using a Merkle Proof and comparing the reconstructed root hash with the original Merkle Root.

3 What to do: Part 1: Creating a Merkle Tree

First, we'll create a Merkle Tree using the crums-io library.

1. Import necessary classes: Begin by importing the required classes from the crums-io library into your Java program. This will likely include `io.crums.util.mrkl.Builder` and others.

2. Initialize the Builder: Create an instance of the `Builder` class.
3. Add data blocks: Add the following three data blocks to your Merkle Tree. Convert these strings to byte arrays before adding them to the builder:
   - Data Block 1: This is the first piece of data.
   - Data Block 2: Here is the second data segment.
   - Data Block 3: And finally, the third part.

   Add each of these data blocks to the `Builder` using the `add()` method.
4. Build the Merkle Tree: Once all data blocks have been added, call the `build()` method on the builder to construct the `Tree` object.
5. Retrieve the Merkle Root: Obtain the Merkle Root of the created tree. Print the Merkle Root (convert the byte array to a hexadecimal string for easier viewing) and total number of leaf nodes.

4 What to do: Part 2: Verifying the Contents of a Merkle Tree

Now we'll verify if a specific data block is part of the Merkle Tree you created in Part 1 and if its integrity has been maintained.

1. Retrieve a data block: Choose one of the original data blocks you added to the tree in Part 1 (e.g., "Here is the second data segment."). Convert the string to a byte array.
2. Generate a Merkle Proof: Using the `Tree` object you created in Part 1, call the proof method with the byte array of the data block you chose in the previous step. This will return a `Proof` representing the Merkle Proof.
3. Verify the data block: Use the static `verify` method to verify the inclusion of the chosen data block.
4. Print the boolean result returned. It should be `true` if the data block is indeed part of the tree and has not been modified.
5. Attempt to verify a modified data block: Take the same data block you used in step 1 and make a small modification to it (e.g., "Here is the second data segment**s**."). Convert the modified string to a byte array.
6. Verify the modified data block: Use the `verify()` method again, this time with the modified data block and the *same* Merkle Proof you generated in step 2. Print the boolean result. It should be `false` because the data has been altered.

